On the Prediction of Java Object Lifetimes

Hajime Inoue, Darko Stefanović, *Member*, *IEEE*, and Stephanie Forrest, *Member*, *IEEE*

Abstract—Accurately predicting object lifetimes is important for improving memory management systems. Current garbage collectors make relatively coarse-grained predictions (e.g., "short-lived" versus "long-lived") and rely on application-independent heuristics related to the local characteristics of an allocation. This paper introduces a prediction method which is fully precise and makes its predictions based on application-specific training rather than application-independent heuristics. By "fully precise" we mean that the granularity of predictions is equal to the smallest unit of allocation. The method described here is the first to combine high precision and efficiency in a single lifetime predictor. Fully precise prediction enables us, for the first time, to study zero-lifetime objects. The paper reports results showing that zero-lifetime objects comprise a significant fraction of object allocations in benchmark programs for the Java programming language and that they are correlated with their allocation context (the call stack and allocation site). Beyond zero-lifetime objects, the paper reports results on predicting longer lived objects, where, in some cases, it is possible to predict the lifetime of objects based on their allocation context (the call stack and allocation site) well. For the SPEC benchmark programs, the number of dynamically allocated objects whose call sites have accurate predictors ranges from 0.2 percent to 61 percent. This method could potentially improve the performance of garbage collectors. The paper proposes a death-ordered collector (DOC) and analyzes its implementation overheads and its best possible performance. The study shows how memory performance could be enhanced using the extra information provided by fully precise prediction.

Index Terms—Object lifetimes, workload characterization, pretenuring, object-oriented programming languages, garbage collection, program behavior.

1 Introduction

Garbage-collected languages, such as C# and Java, are increasingly important. Garbage collection (GC) improves developers' productivity by removing the need for explicit memory reclamation, thereby eliminating a significant source of programming error. However, garbage-collected languages incur increased overhead and, consequently, improvement in their performance is essential to the continuing success of these languages. Many algorithms have been proposed over the several decades since GC was invented, but their performance has been heavily application dependent. For example, Fitzgerald and Tarditi showed that a garbage collector must be tuned to fit a program [1]. Another approach relies on larger heap sizes and simply runs the collection algorithms less frequently. However, this does not always result in better performance [2]. GC algorithms typically make certain assumptions about the lifetimes of the application's objects and tailor the collection algorithm to these assumptions. If the assumptions are not borne out, poor performance is the outcome. What is needed is the ability to make accurate and precise predictions about object lifetimes and to incorporate these predictions into a general GC algorithm that works well for a wide range of applications.

The overhead of GC, compared to explicit deallocation, arises from the cost of identifying which objects are still active (*live*) and which are no longer needed (*dead*). GC algorithms, therefore, go to some lengths to collect regions of memory that are mostly dead. The ideal garbage collector would collect regions where all the objects died recently so that heap space is not wasted by dead objects and living objects are not processed unnecessarily. To do this, the allocator would need to know the exact death time of an object at the time it was allocated and then it could allocate it to a region occupied by objects with the same death time. To date, this has been accomplished only in a limited way by a process called "pretenuring." Pretenuring algorithms make coarse predictions of object lifetimes, predicting which allocations will result in long-lived objects and then allocating them to regions that are not frequently collected. For example, in Blackburn et al.'s pretenuring scheme [3], objects are allocated into short-lived, long-lived, and eternal regions. As this paper will show, the inability to predict lifetimes precisely is an obstacle to the ideal garbage collector.

Modern language runtime environments provide a wealth of profiling mechanisms to investigate the state of an application. In virtual machine (VM) environments such as C# and Java, profiling is an important part of the (JIT) compilation process and may be exploited to improve the performance of all VM components. In this paper, we show how allocation-site information available to the VM can be leveraged to improve object lifetime prediction and how that ability might be exploited by the JIT compiler and collection system.

The organization of this paper is as follows: First, we demonstrate that there is a significant correlation between the state of the stack at an allocation point and the allocated

- H. Inoue is with the School of Computer Science, Carleton University, Herzberg Building, 1125 Colonel By Drive, Ottawa, Ontario, K1S 5B6 Canada. E-mail: hinoue@ccsl.carleton.ca.
- D. Stefanović and S. Forrest are with the Department of Computer Science, MSC01 1130, 1 University of New Mexico, Albuquerque, NM 87131-0001. E-mail: {darko, forrest}@cs.unm.edu.

Manuscript received 19 Dec. 2003; revised 16 Dec. 2004; accepted 7 Dec. 2005; published online 22 May 2006.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0291-1203.

object's lifetime. Next, we describe how this information can be used to predict object lifetimes at the time they are allocated. We then show that a significant proportion of objects have zero lifetime. Next, we analyze the behavior of a hypothetical hybrid GC (the death-ordered collector) that uses our prediction method, examining its implementation overheads and describing its best-case behavior. Finally, we compare our results with related work and discuss other potential applications.

2 Object Lifetime Prediction

Our approach is inspired by Barrett and Zorn's work on object lifetime prediction in C applications [4]. In particular, both methods use similar information, the predictors are constructed similarly using runtime profiling, and we have adopted their "self prediction" test. In addition, we have made several extensions. First, we are using a garbage collected language, Java, in which deallocation is implicit. Second, we have introduced fully precise prediction; Barrett and Zorn used only two bins—short and long-lived. Finally, we have conducted a more detailed analysis, the contents of which form the bulk of this paper.

As mentioned earlier, one goal of object lifetime prediction is to improve performance by providing runtime advice to the memory allocation subsystem about an object's likely lifetime at the time it is allocated. Similarly to Barrett and Zorn, we accomplish this by constructing an object lifetime *predictor*, which bases its predictions on information available at allocation time. This includes the context of the allocation request, namely, the dynamic sequence of method calls that led to the request, and the actual type of the object being allocated. We refer to this information as the *allocation context*; if the observed lifetimes of all objects with the same allocation context are identical, then the predictor should predict that value at runtime for all objects allocated at the site.

We have not yet integrated our predictor into the memory allocation subsystem, so our testing is trace-driven and not performed at runtime. If fully implemented, our system would operate similarly to other profile-driven optimization systems. First, a small training run would be used to generate the predictor, instead of logging the trace. Subsequent runs would then use the predictor for the various optimizations discussed below.

We consider two circumstances for prediction: self prediction and true prediction. *Self prediction* [4] uses the same program trace for training (predictor construction) as for testing (predictor use). Self prediction provides an initial test of the hypothesis that allocation contexts are good predictors of object lifetimes. Although self prediction is not predicting anything new, it allows us to study the extent to which the state of the stack is correlated with the lifetime of the object allocated at that point. This provides evidence that true prediction is possible. *True prediction* is the more realistic case in which one small training trace is used to construct the predictor and a different larger trace (generated from the same program but using different inputs) is used for testing. If self prediction performance is poor, then true prediction is unlikely to succeed. But, accurate self prediction does not necessarily imply successful true prediction. Although we have not analyzed it in detail, we expect that this latter case is most likely to occur in heavily data-driven programs.

The load on the memory-management subsystem is determined by heap allocation and death events and it is independent of other computational effects of the program. Therefore, the lifetime of an object in GC studies is determined by the number of bytes of new memory that are allocated between its birth and its death. More specifically, object lifetime is defined as the sum of the sizes of other objects allocated between the given object's allocation and death and it is expressed in bytes or words.

We evaluate predictor performance using four quantitative measures: precision, coverage, accuracy, and size:

- *Precision* is the granularity of the prediction in bytes. A fully precise predictor has precision of 1 byte, e.g., it might predict that a certain allocation site always yields objects with a lifetime of 10,304 bytes. A less precise predictor might predict from a set of geometrically proportioned bins, such as 8,192-16,383 (we refer to these as *granulated predictors*), or, as we mentioned before, from a small set of bins such as short-lived, long-lived, and eternal. Our aim is to achieve high precision (narrow ranges). In practice, the ideal precision will depend on how the memory allocation subsystem exploits the predictions.
- *Coverage* is the percentage of objects for which the predictor makes a prediction. We construct the predictors so that they make predictions only for allocation contexts that can be predicted with high confidence. Thus, in some cases, the predictor will make no prediction, rather than one that is unlikely to be correct, and the memory allocation subsystem will need a fallback allocation strategy for these cases. Although the decision to predict is made per allocation site, the natural measure of coverage is the percentage of actual object allocation events that are predicted (a dynamic count) rather than the percentage of sites at which a prediction can be made (a static count). Ideally, coverage should be as high as possible.
- *Accuracy* is the percentage of predicted objects which are predicted correctly. That is, among all the objects allocated at runtime for which a prediction is made, some will have a true lifetime that falls in the same range as the predicted lifetime; the range is defined by the precision parameter. Accuracy should be as high as possible.
- *Size* is the number of entries in the predictor, where an entry consists of a descriptor of an allocation site and the prediction for that site. Because the predictor incurs space and time overhead at runtime, smaller sizes are better.

There are interesting trade-offs among precision, coverage, accuracy, and size. For example, a predictor must choose between coverage and precision. Increasing the predictor size (adding more entries) allows either greater coverage (by specifying predictions for objects not previously covered) or greater precision (by specifying

Stack String Prefix	Type	Lifetime
2724:1563:1858:1490:3984:3030	[B	64

Fig. 1. A single predictor entry: The SSP describes the execution path of the program. Each integer encodes the method and position within the method of the next method call. The entire string denotes the allocation site. All byte arrays (JVM type [B) allocated with this stack string prefix had a lifetime of 64 bytes.

different predictions for those objects). There is also the obvious trade-off between coverage and accuracy.

We construct predictors in two stages. First, we collect an execution trace for each program and then we construct the predictor itself. The trace includes accurate records of each object allocation and death. For each allocation event, we record the object identifier, its type, and its execution context. The execution context represents the state of the entire stack at the time of allocation, consisting of the identifiers of the methods and bytecode offsets, stored as integers. We refer to this information as the *stack string*. In most cases, we reduce the amount of information by recording only the top few entries of the stack (the *stack string prefix*, or SSP[1]) and study the effects of varying the length of the prefix. Each death event is recorded to a precision of 1 byte. This full precision is unique in object lifetime traces of garbage-collected languages. Object lifetimes reported in the literature almost always have coarse granularity for garbage-collected languages [3]. This is because object death events can only be detected at a collection point and collections are relatively infrequent. We used an implementation of the Merlin generation trace algorithm [5] within the JikesRVM open-source Java virtual machine [6] to collect fully precise object-lifetime data. Merlin makes absolute precision practical by not enforcing frequent garbage collections. Instead, it imprints a timestamp on objects at memory roots when an allocation occurs.[2] During a collection, an object's death time can be computed by determining the last timestamp in the transitive closure of a group of objects.

A predictor consists of a set of predictor entries. A predictor entry is a three-tuple <SSP, type, lifetime>, as shown in Fig. 1. The trace is used to construct a predictor for the corresponding program. An entry is added to the predictor if and only if all lifetimes during training corresponding to the SSP and type are identical. This implies that if any two objects allocated at the same allocation site have different lifetimes, or *collide*, the predictor will not make a prediction for that allocation site.

This type of predictor is computationally efficient and tunable. For example, *singletons* typically dominate the predictor and increase its size significantly. Singletons are entries for which only one object was allocated during training, so no collisions could have eliminated them. These entries might be removed to form a smaller predictor without greatly reducing coverage (because each entry is used so infrequently), as shown in Fig. 4.[3] More sophisticated strategies could also be devised to optimally balance coverage and size.

1. This is not to be confused with the supervisor stack pointer which has the same acronym.

2. Memory roots are the references to objects that a program can manipulate directly. Examples are registers and, particularly in Java, the references in the program stack.

3. The benchmark *perimeter* from the Java Olden suite [7], [8] is used here because it displays the behavioral archetype.

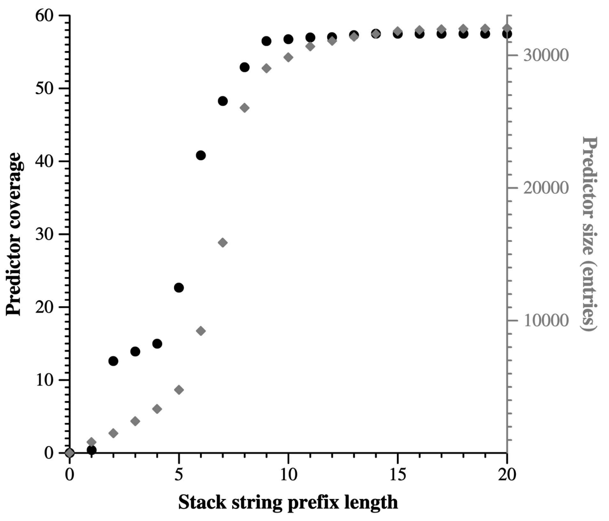

Fig. 2. The effect of stack prefix length on predictor size and coverage for the example benchmark *pseudojbb* (including singletons).

We consider three aspects of lifetime prediction:

- *Fully Precise Lifetime Prediction.* Object lifetimes are predicted to the exact byte.
- *Granulated Lifetime Prediction.* A lower precision approach in which the predictor bins lifetimes according to the scheme $bin = \log_2(lifetime + 1)$. Because most objects die young, the effective precision of this method is still high.
- *Zero Lifetime Prediction.* The predictor predicts only zero lifetime objects (those that die before the next object allocation). We discovered that some benchmarks generate a large number of zero-lifetime objects. Predicting zero-lifetime objects is an interesting subproblem of fully precise prediction.

We illustrate these concepts and tradeoffs on the example benchmark *pseudojbb*.[4] Fig. 2 shows how predictor coverage and size depend on the SSP length as it is varied from 0 to 20; we used fully precise lifetime prediction and singletons were retained in the predictor. Fig. 2 plots the SSP length along the horizontal axis. For each plotted SSP value, we synthesized a predictor from the training trace. The predictor's coverage, i.e., the percentage of object allocations in the trace for which the predictor makes a prediction, is plotted along the vertical axis. Predictor coverage improves with increasing SSP length as more information is available to disambiguate allocation contexts. However, this effect plateaus at an SSP of about length 10, suggesting that 10 is a sufficient SSP length. Fig. 2 also shows the growth of predictor size, i.e., the number of entries, with increasing SSP length.

Fig. 3 allows us to see the effect of removing singletons from the predictor. Notably, predictor coverage is almost unchanged, but predictor size is dramatically reduced. Excluding singletons reveals interesting dependencies

4. Section 3 describes the benchmarks used in our study.

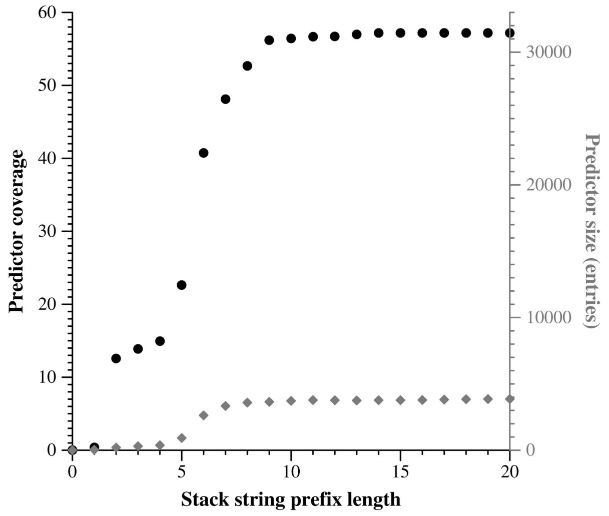

Fig. 3. The effect of stack prefix length on predictor size and coverage for the example benchmark *pseudojbb* (excluding singletons).

among SSP length, predictor size, and coverage. There is a trade-off between collisions and singletons—if the SSP is too short, too many objects collide; if it is too long, the SSP converts the entries into singletons and bloats predictor size. This effect is illustrated by the *perimeter* benchmark, shown in Fig. 4, which has a maximum at around SSP length 8, and then decays to a plateau. The maximum divides the regime of collisions, on the left, and the regime of singletons, to the right.

We report results in object counts, rather than bytes, because we are interested in how well the predictor performs. Object counts are the natural unit for this consideration. However, bytes are often used in the garbage collector literature and we collected these data as well with similar results. In other words, object size is not significantly correlated with the ability to predict object lifetimes.

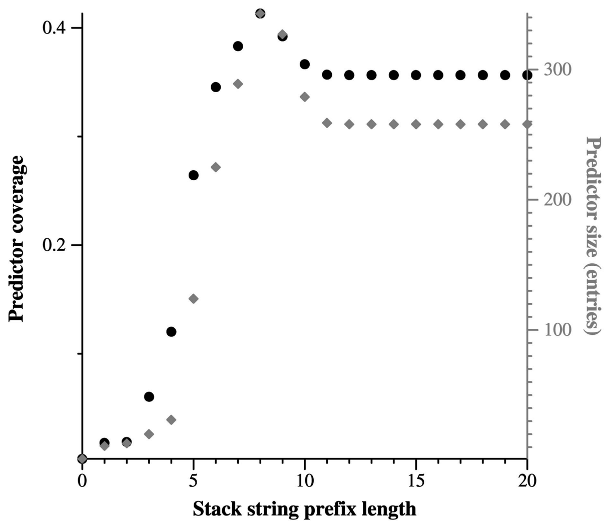

Fig. 4. The effect of stack prefix length on predictor size and coverage for the Java Olden benchmark *perimeter* (excluding singletons).

TABLE 1
The Benchmarks and a Short Description of What They Compute

Benchmark	Description
compress	Uses Lempel-Ziv to compress and decompress some strings
jess	Expert systems shell based on NASA's CLIPS system
db	Emulates database operations on a memory resident database
javac	The Java compiler from jdk 1.0.2
mpegaudio	Decodes mpeg layer3 (mp3) files
mtrt	Multi-threaded raytracer draws a scene with a dinosaur
jack	Java parser generator is a lex and yacc equivalent
pseudojbb	Java Business Benchmark altered for GC research

The first group contains the specjvm98 benchmarks and the second group contains the specjbb2000 benchmark.

3 Benchmarks and Experimental Testbed

We report data on the SPECjvm98 and SPECjbb2000 benchmarks. We also collected data on the Java Olden benchmarks [9] (data not shown), but their small, synthetic nature produces outlier behavior. The SPEC benchmarks consist of useful "real world" programs (with the exception of *db*, a synthetic benchmark constructed to simulate database queries) and are intended to be representative of real applications run on Java virtual machines. They are written in a variety of programming styles and exercise the heap differently. For a detailed study of the individual benchmarks' memory behavior, see Dieckmann and Hölzle [10]. Table 1 describes the individual benchmarks and Table 2 gives some general runtime characteristics of the benchmarks.

We used the JikesRVM Java Virtual Machine version 2.0.3 from IBM. The specific configuration was OptBaseSemispace with the Merlin extensions [5]. This means that the optimizing compiler was used to compile the VM and the baseline compiler compiled the benchmarks. The GC was the default semispace collector (though note that the generated trace is independent of the collector used).

4 Self Prediction

Self prediction tests the predictor using the same trace from which it was constructed. Thus, prediction accuracy is not an issue—if the predictor makes a prediction at all, it will be correct. Of interest are the trade-offs among precision, coverage, and size. We report results for full-precision and logarithmic granularities and we consider the effects of including or excluding singletons from the predictors. Table 3 shows the results. For each benchmark, a set of preliminary runs was conducted to determine the optimal SSP value (shown in the columns labeled "SSP"). The optimal SSP value was determined separately for fully precise and for logarithmic cases.

4.1 Fully Precise Self Prediction

The results for fully precise self prediction (exact granularity) are shown in the left half of Table 3, for predictors with and without singletons.

4.1.1 Predictors Including Singletons

All of the benchmarks show some level of coverage, notably on the synthetic benchmark *db*. Greater than 50 percent

TABLE 2
Trace Statistics

Testing and self prediction				
Benchmark	Command line	Objects allocated	Bytes allocated	Static Allocation sites
compress	-s100	24206	111807704	707
jess	-s100	5971861	203732580	1162
db	-s100	3234616	79433536	719
mpegaudio	-s100	39763	3579980	1867
mtrt	-s100	6538354	147199132	897
javac	-s100	7287024	228438896	1816
raytrace	-s100	6399963	142288572	992
jack	-s100	7150752	288865804	1184
pseudojbb	70000 transactions	8729665	259005968	1634

Training				
Benchmark	Command line	Objects allocated	Bytes allocated	Static sites
jess	-s1	125955	6978524	1143
javac	-s1	20457	2641064	1188
mtrt	-s1	328758	11300528	989
jack	-s1	512401	21911316	1183
pseudojbb	10000 transactions	2778857	103683020	1634

For each trace, the number of objects allocated (Column 3) and the total size of all allocated objects (Column 4) are given. Column 5 shows the number of allocation contexts; each site is counted only once, even if executed more than once, and sites that are not executed in these particular runs are not counted. The top section of the table lists the traces used for the self prediction study (Section 4). The bottom part of the table lists the training traces used in the true prediction study (Section 5); traces from the top section are reused for testing true prediction.

TABLE 3
Self-Prediction Results

Benchmark	Fully Precise					Logarithmic				
	incl. singletons		excluding singletons			incl. singletons		excluding singletons		
	size	coverage	size	coverage	SSP	size	coverage	size	coverage	SSP
compress	9133	67.78	792	33.33	10	9692	85.66	1381	84.84	10
jess	24999	23.40	3326	23.03	24	25873	34.87	4197	34.51	26
db	8847	90.36	73	89.91	3	9474	90.56	348	90.09	4
raytrace	14761	41.61	325	41.27	4	15509	42.36	514	41.97	4
javac	109357	28.63	75571	28.17	32	144844	45.72	111058	45.25	32
mpegaudio	17550	78.42	1931	39.67	8	18260	89.68	2704	51.12	9
mtrt	12490	50.11	1566	49.79	3	13167	50.92	306	50.53	3
jack	29542	61.25	14126	61.04	20	31659	66.30	14600	65.90	17
pseudojbb	32044	57.52	3861	57.20	14	33158	63.65	4861	63.23	14

The first two columns of fully precise and logarithmic granularity give results using predictors including singletons using an SSP of length 20, with two exceptions: jess and javac, for which we used the larger SSP value reported in the fifth column.

coverage is achieved on *compress*, *mpegaudio*, *mtrt*, *pseudojbb*, and *jack* and more than 20 percent on the remaining two, *jess* and *javac*. The predictor achieved greater than 90 percent coverage only on *db*. From our experience with *db* and a set of smaller benchmarks not reported here, we believe that very high coverage numbers are not typical of realistic applications. These results suggest that fully precise prediction can achieve reasonable coverage on some, but not all, applications. As we will see, however, even moderate coverage may be beneficial (Section 8).

4.1.2 Predictors Excluding Singletons

When singleton entries are removed from the predictors (Columns 4-5 in Table 3), we expect coverage to drop, but we would like to know by how much.

Two benchmarks, *compress* and *mpegaudio*, have predictors that include large numbers of singletons. The coverage for each drops roughly by half when singletons are excluded. For the rest, coverage drops by less than a percent, while the number of predictor entries shrinks dramatically: The smallest decrease is a 30 percent drop by *javac*, while *db* is less than 1 percent its previous size. The average decrease in size is greater than 76 percent. Again, *db* could be anomalous because it is synthetic.

Although other studies of lifetimes, such as Harris [11], based prediction only on current method and type, we found no benchmarks for which type alone was sufficient to generate predictors with significant coverage. Benchmarks *jess* and *mtrt* needed at least the method containing the allocation site (an SSP of length one) to have significant coverage and the rest needed more. Perhaps type is not required at all, but it disambiguates allocations in the rare case that different types are allocated at the same allocation point. Note that we are using a flow-sensitive notion of the stack, recording both the method and bytecode offset of each call.

In summary, the fully precise predictors cover a significant fraction of all benchmarks. With singletons excluded, the predictors still have significant coverage while decreasing the number of entries by an average of 76 percent.

4.2 Logarithmic Granularity

As one might expect, the performance of the granulated predictors, also shown in Table 3, is better than the fully precise predictors, because it is an easier problem. Improvement in coverage ranges from *db*'s less than 1 percent to *javac*'s greater than 60 percent. The average is 7 percent improvement. The behavior of the granulated predictors when removing singletons is similar to the fully precise case. The only dramatic change is in *compress*, which behaves like the rest of the benchmarks; *mpegaudio* remains the outlier.

Because of their logarithmic bin size, the predictors are highly precise for the large number of short-lived objects. Granulated prediction has more immediate application than the exact case because the training phase could, in principle, be performed more quickly and without relying on the Merlin algorithm.[5] We concentrate on the exact case, however, because we expect that information about exact behaviors will reveal new avenues for optimization.

4.3 Variations

We studied a broader definition of predictor—one in which the predictor handled lifetimes that varied over each allocation as an arithmetic progression. For example, consider a loop that allocates an object of a particular type to a linked list. When the loop exits, some computation is performed on the list and it is then collected. Each object in the loop has a lifetime that is less than its predecessor by a constant that is the size of the object. To handle this, we need to predict using differences. More formally, the predictor entries become four-tuples <SSP, type, lifetime, increment>. The lifetime is not set during training unless the increment is found to be zero. It is updated every time an object is allocated by adding the increment. In our example, a predictor entry would be created if each object's lifetime differed by a constant increment in the order in which it was allocated. The predictor's lifetime would be initialized by the first object allocated during testing. Subsequent predictions would be made by adding the increment (which would be negative) to the lifetime. Note that absolute lifetime predictions cannot be made online until after the first object matching an entry has died.

Interestingly, this new predictor does not perform well for any of the benchmarks. We found only one benchmark for which it performs well, *em3d* from the Java Olden suite, in which differential allocations account for 36.39 percent of all predictions. This is not a good indicator of predictive strength, however, because almost all of these objects are singletons. For the other benchmarks, the increase in predictive ability averaged 1.1 percent.

Another variation is to use the optimizing compiler both for runtime and the benchmarks (the OptOptSemispace configuration). We tested this variation and the results are qualitatively similar to the earlier experiments (data not shown).

5. This could be accomplished using a generational collector in which collection for a generation is forced at time multiples of its previous generation. During training, for example, the second generation would be collected once for every two collections of the nursery for a logarithmic granularity with the base the size of the nursery. This allows for pretenuring type optimizations, but falls short of the "ideal" garbage collector.

TABLE 4
True Prediction

Benchmark	Full Precision		Logarithmic	
	Coverage	Accuracy	Coverage	Accuracy
jess	1.22	99.88	1.34	99.95
javac	0.48	81.79	0.54	85.95
mtrt	0.18	99.28	0.24	99.77
jack	61.45	99.87	67.17	99.69
pseudojbb	57.09	99.99	63.20	99.85

Coverage and accuracy using predictors generated from a benchmark run using a smaller set of input for both fully precise and logarithmic granularities against a separate, larger benchmark run. Coverage is the percentage of objects for which we make predictions, and accuracy is percentage of those objects for which our predicted lifetime was correct.

5 True Prediction

Barrett and Zorn found that true prediction accuracy is high for those benchmarks that have high coverage in self prediction and are not data-driven. We tested true prediction against a subset of the benchmarks to see if this correlation holds with higher levels of precision. We used *jess*, *javac*, *mtrt*, *jack*, and *pseudojbb*. We used the SSP lengths specified in Table 3 and included singletons. Although this was not an exhaustive study, it demonstrated that true prediction performs well, with results comparable to the Barrett and Zorn study, even with the more stringent requirement of full precision.

Results for the five examples are shown in Table 4 . For both fully precise and logarithmic granularity, all of the predictors are highly accurate. For three of the benchmarks, however, the high accuracy comes at the price of coverage. Coverage is insignificant for *jess*, *javac*, and *mtrt*. The other benchmark predictors show considerable coverage. The difference in coverage is probably due to the degree to which the program is data-driven. For example, the training run of *jess* is quite different from its test run. In *pseudojbb*, the only difference is the length of the run. Although not exhaustive, these examples give evidence that highly precise, true prediction is possible for some applications and that, when precise prediction is possible, it is highly accurate.

6 Zero-Lifetime Objects

A zero-lifetime object is allocated and then dies before the next object is allocated. Our ability to study object lifetimes with full precision allows us to study the behavior of zero-lifetime objects.

Table 5 shows the fraction of zero-lifetime objects generated by each benchmark and the fraction of those that we were able to predict using self prediction. Interestingly, many of the benchmarks allocate large numbers of zero-lifetime objects.

All of the SPEC benchmarks generate a large percentage of zero-lifetime objects, with *javac* allocating the least at 13 percent. We explore the potential consequences of this result in Section 8.

TABLE 5
Fully Precise Zero Lifetime Self Prediction

Benchmark	% of all objects	% predicted	% predicted of possible
compress	21.72	20.86	96.03
jess	39.63	19.96	50.36
db	45.06	45.01	99.97
mpegaudio	25.98	25.29	97.34
mtrt	40.01	33.37	83.39
javac	12.95	10.48	80.93
raytrace	41.30	29.57	71.60
jack	43.44	0.22	0.49
pseudojbb	20.82	18.75	90.04

Column 1 lists the benchmark program, Column 2 shows the fraction of zero-lifetime objects out of all dynamically allocated objects for that benchmark, Column 3 shows the percentage of zero-lifetime objects predicted (coverage), and Column 4 shows the prediction accuracy. SSP lengths are as described in Table 3.

7 Prediction and Object Types

In order to study how prediction results are affected by an object's type, we developed a simple classification of allocated objects according to their type. We used the following categories: application types, library types, and virtual machine types (since the virtual machine we use is written in Java itself). Library types are those classes belonging to the `java` hierarchy. VM classes are easily identified by their `VM` prefix. Application classes are all others.

As Table 6 shows, global coverage (defined as greater than 90 percent) was usually associated with high coverage of application types. This makes sense because application types dominate for most benchmarks. The exceptions, *tsp* and *db*, allocate many library types, which also have high coverage. A predictor's coverage depends on its ability to predict types resulting from application behavior, rather than the underlying mechanisms of the compiler or VM.

8 Exploiting Predictability: Toward an Ideal Collector

In the previous sections, we demonstrated that, for some programs, we can accurately, and with full precision, predict the lifetimes of a large percentage of objects. In this section, we discuss a possible application of this technique: an improved memory management system.

We begin with an analysis of the maximum performance improvement that could be expected. To do this, we make best-case assumptions; for example, assuming perfect accuracy. We finish with an analysis relaxing this assumption, allowing the collector to handle mispredicted lifetimes. Throughout, we ignore training times. We consider training to be part of the development or installation procedure rather than part of normal execution.

8.1 A Limit Study

In Section 1, we discussed the ideal garbage collector. The core idea behind our simulated allocator is to segment the heap into a nearly ideal collector for those objects whose lifetimes are predictable and to use the rest of the heap in the traditional manner. Our combined memory system is a hybrid of a standard collector and our nearly ideal collector. We refer to the combined system as the *death-ordered collector* (DOC). The nearly ideal heap is composed of two subspaces: the *Known-Lifetimes Space* (KLS) and the *Zero-Lifetimes Space* (ZLS). We assume that the heaps are of fixed size and compare against a semispace collector to simplify the analysis.

The ZLS is simply a section of memory large enough to hold the largest object allocated there during a program execution. No accounting overhead is necessary because these objects have zero lifetime. They die before the next allocation, assuring that it is safe to overwrite them. One might assume that these are stack-allocated.

The KLS is more complicated. It is logically arranged as a series of buckets. Each bucket is stamped with its time-to-die and sorted in order of the stamp, from earliest to latest. It is for that reason we refer to this heap as the death-ordered collector. Upon allocation into this heap, the time-to-death is calculated from the predicted lifetime and current time, a bucket is created for the allocation, inserted into the list, and then newly allocated memory is returned to the application. Collections are easier: The collector simply scans the list, returning buckets to the free-list, until it finds a bucket with a time-to-die greater than the current time. The efficiency of the death-ordered-heap is very high under our current assumptions—only allocation is slower due to the prediction during allocation.

TABLE 6
Self Prediction for Three Categories of Objects According to Object Type

Benchmark	VM		Library		Application		% Predicted
	% Alloc.	% Pred.	% Alloc.	% Pred.	% Alloc.	% Total Pred.	
compress	37.97	63.63	11.89	53.41	50.14	74.33	67.78
jess	0.57	78.90	19.69	99.01	79.74	5.25	23.40
db	0.29	61.47	94.83	94.78	4.88	94.78	90.36
mpegaudio	42.48	70.51	10.16	66.57	47.37	88.05	78.42
mtrt	0.19	68.40	1.94	67.01	97.87	49.74	50.11
javac	15.91	5.86	26.54	32.21	57.54	33.28	28.63
raytrace	0.22	67.97	1.03	66.68	98.76	41.29	41.61
jack	3.22	96.94	48.26	42.99	48.52	77.05	61.25
pseudojbb	0.53	73.35	33.19	33.81	66.27	88.43	57.52

For each of the three categories of types (virtual machine, library, application), the percentage of total allocated objects that fall in the category is given, together with the percentage of objects in the category that are predicted. The rightmost column is the overall percentage of objects predicted (corresponding to the first column of Table 3).

TABLE 7
The Bytes Allocated to the Different Heaps and Their Maximum Sizes and with μ, ϵ, and Factor of Improvement Based on a 50MB Heap

Benchmark	Bytes Allocated			Minimum Size			DOC Allocator Results		
	ZLS	KLS	Semispace	ZLS	KLS	Semispace	$\mu \times 10^3$	ε	Improvement
compress	295868	1510568	111916176	4108	524036	8353592	10.32	0.016	1.01
db	23547452	26227396	31590624	4108	1238758	9239776	24.28	0.61	2.51
mpegaudio	525388	2557324	3902044	4108	524452	3213560	10.33	0.44	1.77
mtrt	47714408	25006584	76978460	4108	523500	8547516	10.31	0.49	1.92
jack	94581356	72275544	125578356	4108	524412	3476036	10.32	0.57	2.30
pseudojbb	44643628	64643588	176059172	4108	545820	28401300	10.74	0.38	1.60

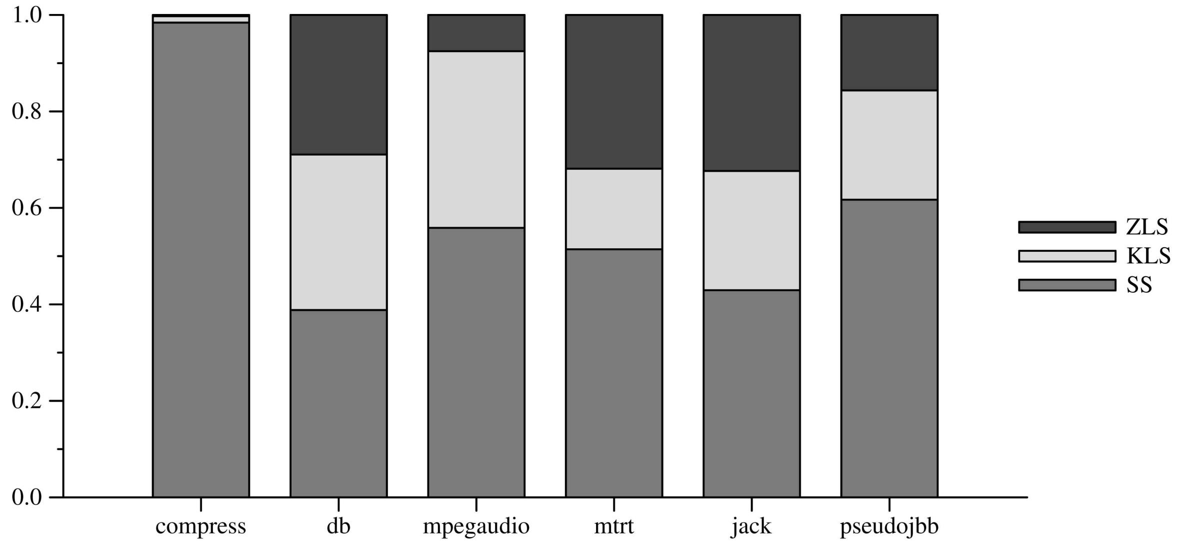

Fig. 5. Death-Ordered Collector: The graph shows the fractional object volume of the different heaps in the simulated benchmarks. ZLS is the Zero-Lifetime Space. SS is the Semispace heap. KLS is the Known-Lifetimes Space.

Whether the hybrid arrangement is efficient depends on the sizes of the heaps and the amount of allocation within each. The sizes of the two heaps depend on their maximum occupancies, which we can measure. Likewise, we know the amount of allocation that would occur in each of the heaps.

The performance of this arrangement thus depends on the allocation characteristics of the application. To study how this would work in practice, we used self prediction to simulate a best-case scenario for several benchmarks that showed a significant (> 50 percent) degree of self prediction. We set the sizes of the ZLS and KLS to the maximum values observed during training. Table 7 provides the absolute numbers of bytes allocated to the three spaces and Fig. 5 shows the relative allocations.

Garbage collector performance can roughly be captured by two metrics: 1) the overall time overhead and 2) the distribution of pause times for collections and time spent in the application between collections. For our DOC system, the time between full collections is the number of bytes allocated before the semispace heap requires collection because the traditional collector dominates ZLS and KLS maintenance. The time between full collections of the SS collector is increased by allocation to the ZLS and KLS heaps, but is reduced due to its smaller size as the total heap size, the combined size of ZLS, KLS, and SS, remains fixed.

We now quantify the performance of the death-ordered collector. Assume h is the total heap size, μ the fraction of the total heap devoted to the KLS and ZLS, ϵ the fraction of bytes allocated into the known and zero lifetimes heaps, and o the heap occupancy after a GC (the survival rate of the heap). The time between collections is simply the number of bytes that are free in a semispace heap after a collection. This is restated as the number of bytes that can be allocated before another collection:

$$T_{DOC} = \frac{(1-\mu)(\frac{h}{2} - \frac{ho}{2})}{1-\epsilon}.$$

The division by 2 comes from the implementation of the semispace collector—a collection occurs when it is half full. The standard case of the single semispace heap occurs when μ and ϵ are 0:

$$T_{ss} = \frac{h}{2} - \frac{ho}{2}.$$

The ratio of the DOC and the semispace equations is the factor of improvement over a single heap:

$$\frac{T_{DOC}}{T_{ss}} = \frac{1-\mu}{1-\epsilon}.$$

TABLE 8
The Ratio of Bytes Copied in the DOC System to the Bytes Copied in the Semispace Collector for Heapsizes of 1.1, 2, and 4 Times the Minimum Semispace Heap Size Required by the DOC System

Benchmark	1.1	2	4
compress	0.84	0.7	0.85
db	0.64	0.35	0.09
mpegaudio	0.51	0.47	0.94
mtrt	0.56	0.49	0.43
jack	0.59	0.43	0.43
pseudojbb	0.63	0.69	0.64

Smaller numbers are preferable.

Therefore, the improvement in time between full collections is simply dependent on ϵ, which can be calculated during the simulation, and μ, which depends upon the chosen heap size. Our results for a heap size of 50MB, Table 7, show improvement for all benchmarks.[6]

We now consider the total time overhead. In copying collectors, like semispace, a good first-order metric is the mark/cons ratio. This is the number of bytes copied by the collector divided by the total number of bytes allocated.

Table 8 shows bytes copied by the DOC heap divided by bytes copied by the single heap. We simulate the heap using sizes of 1.1, 2, and 4 times the minimum size necessary for the hybrid's semispace heap.[7] Here, too, we show improvement for all benchmarks (smaller is better), especially when heap sizes are small. Not only could the DOC achieve a significant reduction in copying cost (40 percent or more), but it would do so across a wide range of heap sizes and for programs in which the baseline overhead of collection is high (mark/cons ratios as high as six, as shown in the "SS" columns of Table 9).

In summary, the DOC heap would both increase the time between allocations and decrease the total pause time. If it could be implemented efficiently, the DOC heap has the potential to greatly increase garbage collection performance.

8.2 More Realistic Implementations

We relax one assumption of the DOC implementation, namely, that it never mispredicts. In the previous section, we showed that DOC is superior to a standard semispace collector if the overhead is small. Here, we argue that the overhead of a realistic implementation is similar to that of a generational collector.

Let us first consider the overhead of allocation. Allocation is more expensive in the DOC because the allocator must decide whether to allocate into the standard heap or the KLS. Placement requires the examination of the SSP, which, in some cases, is quite long. During execution, the SSP would be available only as separate values in individual stack execution frames or perhaps as a separate display-like structure.

6. *compress* shows little improvement, but it is an outlier in terms of memory behavior. It tends to allocate large chunks of memory on startup and only free them on exit.

7. Because the semispace heap reserves half its space at any time, it actually requires twice this amount of memory.

TABLE 9
The Mark/Cons Ratios for Various Heap Sizes of the DOC and Semispace Collector

Benchmark	1.1		2		4	
	DOC	SS	DOC	SS	DOC	SS
compress	0.49	0.59	0.41	0.59	0.09	0.1
db	2.49	3.92	0.25	0.71	0.02	0.25
mpegaudio	0.88	1.72	0.46	0.97	0.46	0.49
mtrt	1.8	3.24	0.29	0.6	0.11	0.25
jack	1.86	3.16	0.33	0.78	0.13	0.29
pseudojbb	3.84	6.1	0.56	0.81	0.18	0.29

To show how allocators incorporating prediction are constructed, we provide an example. At the point of an allocation (a *new* in Java), our compiler would generate inline code for our allocation. Consider this point of execution in the diagram shown in Fig. 6.

The application is at execution point 10 and has three options for allocation. The top two SSPs lead to predictions, the third does not. The object is to compare the current SSP, embodied by the execution stack, with those in the predictor. It is impractical to consider the entire SSP as a single entity because each value within the SSP is a full integer in length. Instead, consider the individual values which make up the SSP: the method ID and position within the method, which is equivalent to the return address. Since the code generated for this allocator will reside within the method, it need not consider any SSP starting with a value other than 10. Nor, since all SSPs we need to consider start with 10, does it need to consider the starting SSP position. Instead, the allocator simply needs to look at enough values of its own stack string to uniquely distinguish between the three SSPs. To accomplish this, the allocator is constructed as a tree. At the first SSP value, it considers values 1 and 3. For SSPs beginning 10:3, there is no prediction, so the compiler generates code for the normal allocation path. For the two SSPs, the compiler then generates code that examines the second value, which makes the prediction of 32,056 bytes for 10:1:2 and 2,016 bytes for 10:1:6. The example is depicted in Fig. 7. Note that these need not be binary trees. For each position, there are as many edges leaving the node as there are unique SSP values at that position. If we assume that each SSP is seen at this allocation point an equal number of times, then the average depth of search is 5/3.

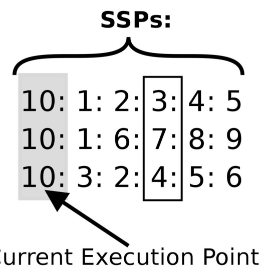

Fig. 6. The three SSPs that need to be considered for an allocation at execution point 10. Although the SSP length is 6, the allocator needs to consider at most two positions for linear comparisons and only one for random access.

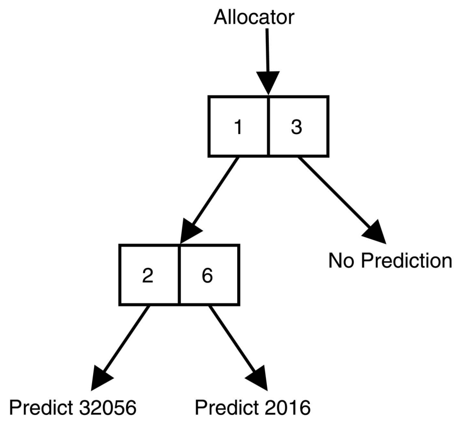

Fig. 7. Linear allocator generated for the execution point 10 shown in Fig. 6. The allocator needs to consider only two positions of the SSP to uniquely distinguish them.

The previous example considers an implementation that assumes that stack examination is similar to walking a list. If a display-like structure is available, where access to any value in the SSP is of constant time (random access), the allocator only needs one node with three values, that for position 3 (outlined in the diagram) and the average depth for search falls to 1. Choosing an optimal sequence of positions when random access is available is, in general, a hard problem. However, the following heuristic works well: Choose the position with the largest number of unique values. This breaks the set down into that number of subsets. Repeat with this heuristic on each subset until no ambiguities exist.

Because all of the stack strings are recorded during the profiling run, the allocator can calculate the average search depth required, as shown in Table 10 for our set of benchmarks. The column labeled "linear" shows results for the algorithm described above that assume that the stack is similar to a list, where execution frames that are further away take longer to examine. However, if we assume constant time random access to the execution frames, we can eliminate many lookups, as shown in the column labeled "random." In each, instead of the full SSP length, for most benchmarks only one to two comparisons are required for each allocation. Even for very long SSP lengths, like those in *jess* and *javac*, the average number of lookups is four or less in the linear case and less than that in the random access case. Thus, allocation can be made efficient in the DOC system.

The overhead of collection in the DOC system is similar to that of a generational collector. Like all generational collectors, the DOC system requires write barriers and remembered sets. The heaps are arranged as described in the previous subsection and shown in Fig. 8. Remembered sets are required between the two heaps, as any multispace collector requires. A remembered set is also required for objects in the KLS pointing to objects with smaller predicted lifetimes. This feature makes the KLS similar to Barret and Zorn's generational collector with a dynamic/threatening boundary [12]. Like that scheme, DOC can collect a variable amount of space to "tune" pause times. It is likely that the remembered sets for objects being collected in the KLS are small, due to our high accuracy. It is difficult to analyze the size of the remembered sets between the standard heap and the KLS without simulation. However, if set size is a problem, one might unify the two heaps into one looking very similar to the Barret and Zorn collector described above, in which our predictions are used as a parameterized pretenuring scheme, with the object lifetime predictions used to determine their placement in the list. This would have overheads very similar to Barret and Zorn's collector. The choice of heap arrangement is an empirical question that involves the relative performance of various parts of the memory management system.

TABLE 10
The Average Number of Hash Table Queries that Are Required to Identify an Allocation Context for a Given SSP and Benchmark

Benchmark	ssp	linear	random	ssp	linear	random
compress	20	1.92	1.28	10	1.92	1.28
jess	24	3.08	2.12	26	3.08	2.12
db	20	1.86	0.98	3	1.85	0.96
raytrace	20	2.00	1.45	4	1.62	1.07
javac	32	3.84	3.27	32	3.84	3.27
mpegaudio	20	2.09	1.40	8	2.06	1.37
mtrt	20	2.01	1.46	3	1.25	0.99
jack	20	4.02	3.21	20	4.02	3.21
pseudojbb	20	2.50	1.89	14	2.48	1.86

The two sets of data shown are for the SSP lengths used in Table 3. For each set, we calculate the average depth using hash tables constructed using the SSP with linear access and with random access.

In the worst case, the DOC's overhead will be similar to that of a standard heap because its overhead is similar to that of a generational collector. When the DOC can use its KLS, then it will outperform a traditional collector. Implementing the DOC within Jikes and measuring its performance experimentally is an area of future investigation.

9 Related Work

There has been little study of Java's memory behavior outside the context of GC algorithms. The focus has been on studying collectors rather than how Java applications use memory. And, if we restrict ourselves to object lifetime prediction, there has been only a small amount of work for any language.

In one of the few studies of Java's allocation behavior, Dieckmann and Hölzle studied in detail the memory behavior of the SPECjvm98 benchmarks using a heap simulator [13], [10]. They found that more than 50 percent of the heap was used by nonreferences (primitives) and that alignment and extra header words expanded heap size significantly since objects tended to be small. They confirmed the weak generational hypothesis for Java, though not as firmly as in other languages (up to 21 percent of all objects were still alive after 1 MB of allocation). This is the most in-depth study of the benchmarks' allocation and lifetime behavior, although a study of access behavior was reported by Shuf et al. [14]. Focusing on garbage collectors, Fitzgerald and Tarditi [1] demonstrated that memory allocation behavior differs dramatically over a variety of Java benchmarks. They pointed out that performance

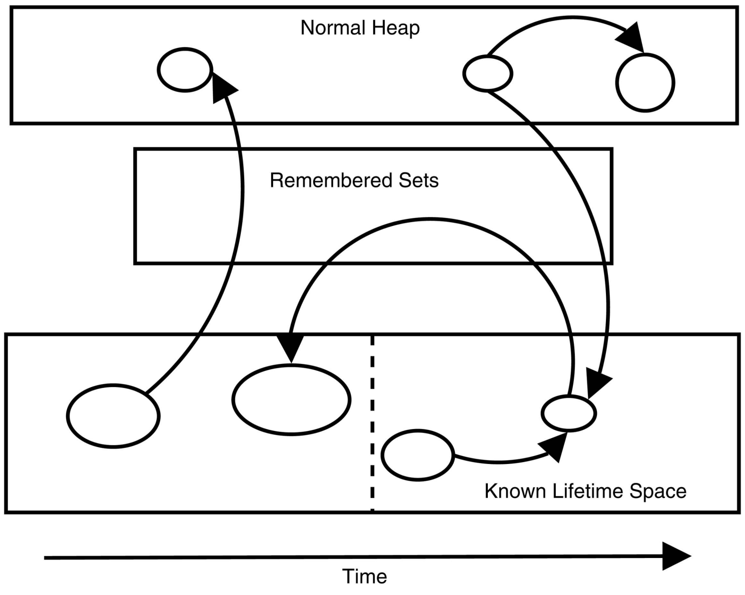

Fig. 8. A visual representation of the death-ordered collector. The ovals within the top and bottom rectangles represent objects within the normal heap and the known lifetime space (KLS), respectively. Objects in the KLS are arranged in order of predicted death. Arrows indicate references from one object to another. Arrows that overlap the remembered sets rectangle must be considered as roots when a collection occurs (though the sets do not have to be unified in any particular implementation). The dotted line indicates the current time. A DOC collection begins with a scan of objects at the left, the ones with the earliest predicted time-of-death, and then moves right, up to the current time. Remembered sets are necessary because objects not considered for collection, those in the normal heap and those not predicted to be dead, are assumed to be alive and, thus, any objects pointed to by them must also be considered alive.

would have improved by at least 15 percent if they had chosen the appropriate collector for the appropriate benchmark. They report that the most important choice is whether or not to use a generational collector and pay the associated penalty for the write barrier.

Lifetime prediction has almost always been studied in the context of pretenuring or similar schemes. These schemes rely on a training or profiling stage to learn lifetimes before they can be exploited. Static heuristics have not been used in published work to this point, although Jump and Hardekopf found that objects that escape their thread are usually long-lived [15].

Cheng et al. [16] describe pretenuring in an ML compiler using a simple algorithm that associates allocation contexts with lifetime during profiling—sites that produce objects that survive one minor collection with 80 percent probability are pretenured.

As discussed earlier, our work has many similarities to Barrett and Zorn's "Using Lifetime Predictors to Improve Memory Allocation Performance" [4], which used a similar method to construct predictors. However, Barrett and Zorn used C applications, so lifetimes were explicit. Their predictor also was binary; it predicted that objects were either short-lived or long-lived.

Cohn and Singh [17] revisited the results of Barrett and Zorn using decision trees based on the top n words of the stack, which includes function arguments, to classify short-lived and long-lived objects. They improved on Barrett and Zorn's results, but at the cost of computational expense because they used all the stack information. By contrast, our algorithm uses only the method identifier and bytecode offset.

Blackburn et al. [3] used coarse-grained prediction with three categories in Java, using the allocation site and lifetime as features to construct pretenuring advice for garbage collectors. They found they were able to reduce garbage collection times for several types of garbage collection algorithms.

Shuf et al. [18] decided that segregating objects by type rather than age, as in generational collection, was more promising. They found that object types that are allocated most frequently have short lifetimes. They then used the type as a prediction of short lifetime, dividing the heap into "prolific" (or short lifetime) and regular regions.

Seidl and Zorn [19], [20] sought to predict objects according to four categories: highly referenced, short-lived, low referenced, and other. Their goal was to improve virtual memory behavior rather than cache performance as in [4]. Their prediction scheme was again based on the stack; they emphasized that, during profiling, it was important to choose the right depth of the stack predictor: Too shallow is not predictive enough and too deep results in overspecialization.

Harris [11] studied pretenuring using only the current method signature and bytecode offset. He considered using the SSP, but decided it provided little information unless recursion is removed. He speculated that using the class hierarchy might be an easier and less expensive way to predict lifetimes as related types usually have the same lifetime characteristics. His conclusion about the usefulness of SSP may have been a result of his methodology (he considered a maximum SSP length of 5) and the large granularity (short and long-lived objects).

Some studies used more information than just the stack and allocation site. These typically do not do pretenuring, which concentrates on where to put an allocation and, therefore, needs a lifetime prediction at birth. Rather, these other methods focused on finding an efficient time to collect and, thus, made relative predictions about deaths.

Cannarozzi et al. [21] used a single-threaded program model and kept track of the last stack frame that referenced an object. They observed that, when the last reference is popped, objects in that frame are likely to be garbage.

For example, Hayes [22], [23], using simulation, examined which objects were entry or "key" objects into clusters of objects that die when the keyed object dies. For automatically choosing what objects are keyed, he suggested random selection, monitoring the stack for when pointers are popped, creating key objects, and doing processing during promotion in generational garbage collection. In effect, the keyed objects are used to sample the clusters.

Similarly, Hirzel et al. [24] looked at connectivity in the heap to discover correlations among object lifetimes. They found that objects accessible from the stack have short lifetimes, objects accessible from globals are very long-lived, and objects connected via pointers usually die at about the same time.

Our own work resembles much of the work described here in its use of the allocation site and stack for constructing the predictor and in its reliance on a training (profiling) phase. Our work extends this earlier work by increasing the precision of lifetime prediction, specifically the ability to make fully precise predictions. In addition, many of the earlier methods do not make specific predictions; indeed, some do not make predictions at all.

One most obvious application of our method is as a hinting system, which would identify objects that might be allocated on the stack instead of the heap. Stack allocation is cheaper than heap allocation and has no garbage collector overhead. Currently, the principal method of identifying such objects is through escape analysis. Determining which objects escape the stack is, in general, a difficult, costly analysis.[8] Although this can be performed statically, in Java it occurs at runtime because the class file format has no way to encode this information. Therefore, it would be helpful to speed up the analysis by identifying objects that are likely to not escape the stack.

10 Discussion and Conclusions

Most GC algorithms are effective when their assumptions about lifetimes match the actual behavior of the applications, but, beyond coarse-grained predictions such as pretenuring, they do little to "tune" themselves to applications. The ideal garbage collector would know the lifetime of every object at its birth. In this paper, we have taken a step toward this goal by showing that, for some applications, it is feasible to predict object lifetimes to the byte (referred to as *fully precise* prediction). In addition, we showed how a memory system could exploit this information to improve its performance.

8. A description of the performance costs of escape analysis can be found in Deutsch's *On the Complexity of Escape Analysis* [25] and a Java specific implementation in Choi's *Escape Analysis for Java* [26].

It is remarkable that fully precise prediction works at all. Previous attempts at prediction used a much larger granularity, in the thousands of bytes. In particular, Barrett and Zorn used a two-class predictor with a division at the age of 32KB. It is not surprising that the predictor they described worked well, given that 75 percent of all objects lived to less than that age. Cohn and Singh's decision trees [17] worked very well at the cost of much greater computational complexity. Blackburn et al.'s pretenuring scheme [3] used a coarse granularity. The method described here is the first to attempt both high precision and efficient lifetime prediction and it does so using a surprisingly simple approach. An area of future investigation is to consider other prediction heuristics and to test them on fully precise prediction. Because our accuracy is already so high, the goal here would be to increase coverage.

Our results show that a significant percentage of all objects live for zero bytes, a result that required the use of exact traces. Because our predictors are able to cover zero-lifetime allocation contexts, the zero-lifetime results have clear applications in code optimization. Zero-lifetime object prediction could be used to guide stack escape analysis so that some objects are allocated on the stack instead of on the heap.

Object lifetime prediction could also be used as a hinting system, both for *where* an allocator should place an object and *when* the garbage collector should try to collect it. This would be a more general procedure than pretenuring and it would support more sophisticated garbage collection algorithms, such as multiple-generation collectors and the Beltway collector [27].

Acknowledgments

The authors are grateful to Matthew Hertz for creating and providing to them the Merlin algorithm implementation. They thank Kathryn McKinley, Amer Diwan, Dennis Chao, and the members of the Adaptive Computation Laboratory at the University of New Mexico for valuable comments on the text of the paper. This material is based upon work supported by the US National Science Foundation (grants ANIR-9986555, CCR-0219587, CCR-0085792, EIZ-0218262, and EIA-0238027), the US Defense Advanced Projects Agency (grant AGR F30602-00-2-0584), Sandia National Laboratories, Microsoft Research, the Intel Corporation, Hewlett-Packard (gift 88425.1), and the Santa Fe Institute. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] R.P. Fitzgerald and D. Tarditi, "The Case for Profile-Directed Selection of Garbage Collectors," *Proc. Second Int'l Symp. Memory Management (ISMM),* pp. 111-120, 2000, citeseer.nj.nec.com/356203. html.

[2] T. Brecht, E. Arjomandi, C. Li, and H. Pham, "Controlling Garbage Collection and Heap Growth to Reduce the Execution Time of Java Applications," *Proc. Object-Oriented Programming, Systems, Langages and Applications Conf. (OOPSLA),* pp. 353-366, 2001, citeseer.nj.nec.com/article/brecht01controlling.html.

[3] S.M. Blackburn, S. Singhai, M. Hertz, K.S. McKinley, and J.E. B. Moss, "Pretenuring for Java," *Proc. SIGPLAN 2001 Conf. Object-Oriented Programming, Languages, and Applications,* pp. 342-352, Oct. 2001.

[4] D.A. Barrett and B.G. Zorn, "Using Lifetime Predictors to Improve Memory Allocation Performance," *Proc. SIGPLAN Conf. Programming Language Design and Implementation,* pp. 187-196, 1993, citeseer.nj.nec.com/barrett93using.html.

[5] M. Hertz, S.M. Blackburn, J.E.B. Moss, K.S. McKinley, and D. Stefanović, "Error-Free Garbage Collection Traces: How to Cheat and Not Get Caught," *Proc. SIGMETRICS 2002 Int'l Conf. Measurement and Modeling of Computer Systems,* pp. 140-151, June 2002.

[6] B. Alpern, D. Attanasio, J. Barton, M. Burke, P. Cheng, J. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, S. Hummel, D. Lieber, V. Litvinov, T. Ngo, M. Mergen, V. Sarkar, M. Serrano, J. Shepherd, S. Smith, V. Sreedhar, H. Srinivasan, and J. Whaley, "The Jalapeño Virtual Machine," *IBM Systems J.,* vol. 39, no. 1, Feb. 2000.

[7] M.C. Carlisle and A. Rogers, "Software Caching and Computation Migration in Olden," *Proc. Fifth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* pp. 29-38, July 1995.

[8] A. Rogers, M. Carlisle, J.H. Reppy, and L.J. Hendren, "Supporting Dynamic Data Structures on Distributed-Memory Machines," *ACM Trans. Programming Languages and Systems,* vol. 17, no. 2, pp. 233-263, Mar. 1995.

[9] B. Cahoon and K. McKinley, "Tolerating Latency by Prefetching Java Objects," *Proc. Workshop Hardware Support for Objects and Microarchitectures for Java,* Oct. 1999, citeseer.nj.nec.com/cahoon99tolerating.html.

[10] S. Dieckmann and U. Hölzle, "The Allocation Behavior of the SPECjvm98 Java Benchmarks," *Performance Evaluation and Benchmarking with Realistic Applications,* R. Eigenman, ed., MIT Press, 2001.

[11] T.L. Harris, "Dynamic Adaptive Pre-Tenuring," *Proc. Second Int'l Symp. Memory Management (ISMM),* pp. 127-136, 2000, citeseer.nj.nec.com/harris00dynamic.html.

[12] D.A. Barrett and B. Zorn, "Garbage Collection Using a Dynamic Threatening Boundary," *Proc. SIGPLAN '95 Conf. Programming Languages Design and Implementation,* pp. 301-314, June 1995.

[13] S. Dieckman and U. Hölzle, "A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks," *Proc. 12th European Conf. Object-Oriented Programming (ECOOP '98),* E. Jul, ed., pp. 92-115, July 1998.

[14] Y. Shuf, M.J. Serrano, M. Gupta, and J.P. Singh, "Characterizing the Memory Behavior of Java Workloads: A Structured View and Opportunities for Optimizations," *SIGMETRICS/Performance,* pp. 194-205, 2001, citeseer.nj.nec.com/shuf00characterizing.html.

[15] M. Jump and B. Hardekopf, "Pretenuring Based on Escape Analysis," Technical Report TR-03-48, Univ. of Texas at Austin, Nov. 2003.

[16] P. Cheng, R. Harper, and P. Lee, "Generational Stack Collection and Profile-Driven Pretenuring," *Proc. SIGPLAN '98 Conf. Programming Languages Design and Implementation,* pp. 162-173, June 1998.

[17] D.A. Cohn and S. Singh, "Predicting Lifetimes in Dynamically Allocated Memory," *Advances in Neural Information Processing Systems,* M.C. Mozer, M.I. Jordan, and T. Petsche, eds., vol. 9, p. 939, MIT Press, 1997, citeseer.nj.nec.com/cohn96predicting.html.

[18] Y. Shuf, M. Gupta, R. Bordawekar, and J.P. Singh, "Exploiting Prolific Types for Memory Management and Optimizations," *Proc. Symp. Principles of Programming Languages,* pp. 295-306, 2002, citeseer.nj.nec.com/shuf02exploiting.html.

[19] M.L. Seidl and B. Zorn, "Predicting References to Dynamically Allocated Objects," Technical Report CU-CS-826-97, Univ. of Colorado, 1997, citeseer.nj.nec.com/seidl97predicting.html.

[20] M.L. Seidl and B.G. Zorn, "Segregating Heap Objects by Reference Behavior and Lifetime," *Proc. Eighth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 12-23, 1998.

[21] D.J. Cannarozzi, M.P. Plezbert, and R.K. Cytron, "Contaminated Garbage Collection," *Proc. 2000 ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI),* pp. 264-273, 2000.

[22] B. Hayes, "Using Key Object Opportunism to Collect Old Objects," *Proc. SIGPLAN 1991 Conf. Object-Oriented Programming, Languages, and Applications,* pp. 33-40, Oct. 1991.

[23] B. Hayes, "Key Objects in Garbage Collection," PhD dissertation, Stanford Univ., Stanford, Calif., Mar. 1993.

[24] M. Hirzel, J. Henkel, A. Diwan, and M. Hind, "Understanding the Connectivity of Heap Objects," *Proc. Third Int'l Symp. Memory Management (ISMM),* pp. 36-49, 2002.

[25] A. Deutsch, "On the Complexity of Escape Analysis," *Proc. 24th Ann. ACM Symp. Principles of Programming Languages,* pp. 358-371, 1997.

[26] J.-D. Choi, M. Gupta, M. Serrano, V.C. Sreedhar, and S. Midkiff, "Escape Analysis for Java," *Proc. 1999 ACM SIGPLAN Conf. Object Oriented Programming Systems, Languages, and Applications (OOPSLA 99),* Nov. 1999.

[27] S.M. Blackburn, R.E. Jones, K.S. McKinley, and J.E.B. Moss, "Beltway: Getting around Garbage Collection Gridlock," *Proc. SIGPLAN 2002 Conf. Programming Languages Design and Implementation (PLDI '02),* June 2002, citeseer.nj.nec.com/blackburn02beltway.html.

Hajime Inoue received the BS degree in biophysics from the University of Michigan and the PhD degree in computer science from the University of New Mexico. His research interests include security in virtual machines and programming language design and implementation. He is a postdoctoral fellow with Carleton University's School of Computer Science.

Darko Stefanović received the Dipl.Ing. degree in electrical engineering from the University of Belgrade and the MS and PhD degrees in computer science from the University of Massachusetts. He is an assistant professor of computer science at the University of New Mexico. His research interests include programming languages, virtual machines, memory management, computer security, and biomolecular computing. He is a member of the IEEE.

Stephanie Forrest received the BA degree from St. John's College, Sante Fe, New Mexico, and the MS and PhD degrees from the University of Michigan, Ann Arbor. She is currently a professor of computer science at the University of New Mexico (UNM), Albuquerque, and a member of the residential faculty at the Sante Fe Institute. Before joining UNM, she was with Teknowledge Inc, Palo Alto, California, and was a Director's Fellow at the Center for Nonlinear Studies, Los Alamos National Laboratory, Los Alamos, New Mexico. Her research interests are in adaptive systems, including genetic algorithms, computational immunology, biological modeling, and computer security. She is a member of the IEEE.